**Readme file**

Requirements to run the code:

python 3

dependencies:

flask

tqdm

flask-cors

Step1:

create a virtual space by the following commands

py -3 -m venv .venv

.venv\Scripts\activate

Step2:

install flask and other dependencies by these commands:

pip install flask

pip install tqdm

pip install flask-cors

Step3:

Now run the flask server by writing the following command

flask run

Step4:

open the ui.html file using any browser

Chose any string from the heuristic.json file and enter in the text box

Click submit and within a few minutes the solved cube along with moves will be displayed on your browser

**App.py**

# from flask import Flask

from flask import Flask, render\_template, request, url\_for, jsonify

import json

import os.path

from cube import RubiksCube

from solver import IDA\_star, build\_heuristic\_db

from flask\_cors import CORS, cross\_origin

app = Flask(\_\_name\_\_)

CORS(app)

@app.route("/", methods=['POST'])

@cross\_origin()

def hello\_world():

    input\_json = request.get\_json(force=True)

    # force=True, above, is necessary if another developer

    # forgot to set the MIME type to 'application/json'

    print ('data from client:', input\_json)

    print(input\_json["string"])

    MAX\_MOVES = 5

    NEW\_HEURISTICS = False

    HEURISTIC\_FILE = 'heuristic.json'

#--------------------------------

    cube = RubiksCube(n=3, state=input\_json["string"])

#--------------------------------

    if os.path.exists(HEURISTIC\_FILE):

        with open(HEURISTIC\_FILE) as f:

            h\_db = json.load(f)

    else:

        h\_db = None

    if h\_db is None or NEW\_HEURISTICS is True:

        actions = [(r, n, d) for r in ['h', 'v', 's'] for d in [0, 1] for n in range(cube.n)]

        h\_db = build\_heuristic\_db(

            cube.stringify(),

            actions,

            max\_moves = MAX\_MOVES,

            heuristic = h\_db

        )

        with open(HEURISTIC\_FILE, 'w', encoding='utf-8') as f:

            json.dump(

                h\_db,

                f,

                ensure\_ascii=False,

                indent=4

            )

    #--------------------------------

    # cube.shuffle(

    #     l\_rot = MAX\_MOVES if MAX\_MOVES < 5 else 5,

    #     u\_rot = MAX\_MOVES

    # )

    cube.show()

    print('----------')

    #--------------------------------

    solver = IDA\_star(h\_db)

    moves = solver.run(cube.stringify())

    arr = []

    arr.append(cube.stringify())

    for m in moves:

        if m[0] == 'h':

            cube.horizontal\_twist(m[1], m[2])

        elif m[0] == 'v':

            cube.vertical\_twist(m[1], m[2])

        elif m[0] == 's':

            cube.side\_twist(m[1], m[2])

        print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

        print(m)

        print(cube.stringify())

        cube.show()

        arr.append(cube.stringify())

    dict = {"states": arr, "moves": moves}

    print(arr)

    print(moves)

    print(dict)

    return dict

@app.route('/test', methods=['POST'])

def my\_test\_endpoint():

    input\_json = request.get\_json(force=True)

    # force=True, above, is necessary if another developer

    # forgot to set the MIME type to 'application/json'

    print ('data from client:', input\_json)

    dictToReturn = {'answer':42}

    return jsonify(dictToReturn)

if \_\_name\_\_ == "\_\_app\_\_":

    app.run(host='0.0.0.0')

**cube.py**

from random import randint, choice

class RubiksCube:

    """

    Class containing the rubiks cube code

    """

    def \_\_init\_\_(

        self,

        n = 3,

        colours = ['w', 'o', 'g', 'r', 'b', 'y'],

        state = None

    ):

        """

        Input: n - integer representing the width and height of the rubiks cube

               colours - list containing the first letter of ever colour you wish to use (Default = ['w', 'o', 'g', 'r', 'b', 'y']) [OPTIONAL]

               state - string representing the current state of the rubix cube (Default = None) [OPTIONAL]

        Description: Initialize the rubiks cube

        Output: None

        """

        if state is None:

            self.n = n

            self.colours = colours

            self.reset()

        else:

            self.n = int((len(state) / 6) \*\* (.5))

            self.colours = []

            self.cube = [[[]]]

            for i, s in enumerate(state):

                if s not in self.colours: self.colours.append(s)

                self.cube[-1][-1].append(s)

                if len(self.cube[-1][-1]) == self.n and len(self.cube[-1]) < self.n:

                    self.cube[-1].append([])

                elif len(self.cube[-1][-1]) == self.n and len(self.cube[-1]) == self.n and i < len(state) - 1:

                    self.cube.append([[]])

    def reset(self):

        """

        Input: None

        Description: Reset the cube to its inital state

        Output: None

        """

        self.cube = [[[c for x in range(self.n)] for y in range(self.n)] for c in self.colours]

    def solved(self):

        """

        Input: None

        Description: Determine if the cube is solved or not

        Output: boolean representing if the cube is solved or not

        """

        for side in self.cube:

            hold = []

            check = True

            for row in side:

                if len(set(row)) == 1:

                    hold.append(row[0])

                else:

                    check = False

                    break

            if check == False:

                break

            if len(set(hold)) > 1:

                check = False

                break

        return check

    def stringify(self):

        """

        Input: None

        Description: Create string representation of the current state of the cube

        Output: string representing the cube current state

        """

        return ''.join([i for r in self.cube for s in r for i in s])

    def shuffle(self, l\_rot = 5, u\_rot = 100):

        """

        Input: l\_rot - integer representing the lower bounds of amount of moves (Default = 5) [OPTIONAL]

               u\_rot - integer representing the upper bounds of amount of moves (Default = 100) [OPTIONAL]

        Description: Shuffles rubiks cube to random solvable state

        Output: None

        """

        moves = randint(l\_rot, u\_rot)

        actions = [

            ('h', 0),

            ('h', 1),

            ('v', 0),

            ('v', 1),

            ('s', 0),

            ('s', 1)

        ]

        for i in range(moves):

            a = choice(actions)

            j = randint(0, self.n - 1)

            if a[0] == 'h':

                self.horizontal\_twist(j, a[1])

            elif a[0] == 'v':

                self.vertical\_twist(j, a[1])

            elif a[0] == 's':

                self.side\_twist(j, a[1])

    def show(self):

        """

        Input: None

        Description: Show the rubiks cube

        Output: None

        """

        spacing = f'{" " \* (len(str(self.cube[0][0])) + 2)}'

        l1 = '\n'.join(spacing + str(c) for c in self.cube[0])

        l2 = '\n'.join('  '.join(str(self.cube[i][j]) for i in range(1,5)) for j in range(len(self.cube[0])))

        l3 = '\n'.join(spacing + str(c) for c in self.cube[5])

        print(f'{l1}\n\n{l2}\n\n{l3}')

    def horizontal\_twist(self, row, direction):

        """

        Input: row - integer representing which row you would like to twist

               direction - boolean representing if you want to twist right or left [left - 0, right - 1]

        Description: Twist desired row of rubiks cube

        Output: None

        """

        if row < len(self.cube[0]):

            if direction == 0: #Twist left

                self.cube[1][row], self.cube[2][row], self.cube[3][row], self.cube[4][row] = (self.cube[2][row],

                                                                                              self.cube[3][row],

                                                                                              self.cube[4][row],

                                                                                              self.cube[1][row])

            elif direction == 1: #Twist right

                self.cube[1][row], self.cube[2][row], self.cube[3][row], self.cube[4][row] = (self.cube[4][row],

                                                                                              self.cube[1][row],

                                                                                              self.cube[2][row],

                                                                                              self.cube[3][row])

            else:

                print(f'ERROR - direction must be 0 (left) or 1 (right)')

                return

            #Rotating connected face

            if direction == 0: #Twist left

                if row == 0:

                    self.cube[0] = [list(x) for x in zip(\*reversed(self.cube[0]))] #Transpose top

                elif row == len(self.cube[0]) - 1:

                    self.cube[5] = [list(x) for x in zip(\*reversed(self.cube[5]))] #Transpose bottom

            elif direction == 1: #Twist right

                if row == 0:

                    self.cube[0] = [list(x) for x in zip(\*self.cube[0])][::-1] #Transpose top

                elif row == len(self.cube[0]) - 1:

                    self.cube[5] = [list(x) for x in zip(\*self.cube[5])][::-1] #Transpose bottom

        else:

            print(f'ERROR - desired row outside of rubiks cube range. Please select a row between 0-{len(self.cube[0])-1}')

            return

    def vertical\_twist(self, column, direction):

        """

        Input: column - integer representing which column you would like to twist

               direction - boolean representing if you want to twist up or down [down - 0, up - 1]

        Description: Twist desired column of rubiks cube

        Output: None

        """

        if column < len(self.cube[0]):

            for i in range(len(self.cube[0])):

                if direction == 0: #Twist down

                    self.cube[0][i][column], self.cube[2][i][column], self.cube[4][-i-1][-column-1], self.cube[5][i][column] = (self.cube[4][-i-1][-column-1],

                                                                                                                                self.cube[0][i][column],

                                                                                                                                self.cube[5][i][column],

                                                                                                                                self.cube[2][i][column])

                elif direction == 1: #Twist up

                    self.cube[0][i][column], self.cube[2][i][column], self.cube[4][-i-1][-column-1], self.cube[5][i][column] = (self.cube[2][i][column],

                                                                                                                                self.cube[5][i][column],

                                                                                                                                self.cube[0][i][column],

                                                                                                                                self.cube[4][-i-1][-column-1])

                else:

                    print(f'ERROR - direction must be 0 (down) or 1 (up)')

                    return

            #Rotating connected face

            if direction == 0: #Twist down

                if column == 0:

                    self.cube[1] = [list(x) for x in zip(\*self.cube[1])][::-1] #Transpose left

                elif column == len(self.cube[0]) - 1:

                    self.cube[3] = [list(x) for x in zip(\*self.cube[3])][::-1] #Transpose right

            elif direction == 1: #Twist up

                if column == 0:

                    self.cube[1] = [list(x) for x in zip(\*reversed(self.cube[1]))] #Transpose left

                elif column == len(self.cube[0]) - 1:

                    self.cube[3] = [list(x) for x in zip(\*reversed(self.cube[3]))] #Transpose right

        else:

            print(f'ERROR - desired column outside of rubiks cube range. Please select a column between 0-{len(self.cube[0])-1}')

            return

    def side\_twist(self, column, direction):

        """

        Input: column - integer representing which column you would like to twist

               direction - boolean representing if you want to twist up or down [down - 0, up - 1]

        Description: Twist desired side column of rubiks cube

        Output: None

        """

        if column < len(self.cube[0]):

            for i in range(len(self.cube[0])):

                if direction == 0: #Twist down

                    self.cube[0][column][i], self.cube[1][-i-1][column], self.cube[3][i][-column-1], self.cube[5][-column-1][-1-i] = (self.cube[3][i][-column-1],

                                                                                                                                      self.cube[0][column][i],

                                                                                                                                      self.cube[5][-column-1][-1-i],

                                                                                                                                      self.cube[1][-i-1][column])

                elif direction == 1: #Twist up

                    self.cube[0][column][i], self.cube[1][-i-1][column], self.cube[3][i][-column-1], self.cube[5][-column-1][-1-i] = (self.cube[1][-i-1][column],

                                                                                                                                      self.cube[5][-column-1][-1-i],

                                                                                                                                      self.cube[0][column][i],

                                                                                                                                      self.cube[3][i][-column-1])

                else:

                    print(f'ERROR - direction must be 0 (down) or 1 (up)')

                    return

            #Rotating connected face

            if direction == 0: #Twist down

                if column == 0:

                    self.cube[4] = [list(x) for x in zip(\*reversed(self.cube[4]))] #Transpose back

                elif column == len(self.cube[0]) - 1:

                    self.cube[2] = [list(x) for x in zip(\*reversed(self.cube[2]))] #Transpose top

            elif direction == 1: #Twist up

                if column == 0:

                    self.cube[4] = [list(x) for x in zip(\*self.cube[4])][::-1] #Transpose back

                elif column == len(self.cube[0]) - 1:

                    self.cube[2] = [list(x) for x in zip(\*self.cube[2])][::-1] #Transpose top

        else:

            print(f'ERROR - desired column outside of rubiks cube range. Please select a column between 0-{len(self.cube[0])-1}')

            return

**solver.py**

from random import choice

from tqdm import tqdm

from cube import RubiksCube

class IDA\_star(object):

    def \_\_init\_\_(self, heuristic, max\_depth = 20):

        """

        Input: heuristic - dictionary representing the heuristic pre computed map

               max\_depth - integer representing the max depth you want your game tree to reach (default = 20) [OPTIONAL]

        Description: Initialize the solver

        Output: None

        """

        self.max\_depth = max\_depth

        self.threshold = max\_depth

        self.min\_threshold = None

        self.heuristic = heuristic

        self.moves = []

    def run(self, state):

        """

        Input: state - string representing the current state of the cube

        Description: solve the rubix cube

        Output: list containing the moves taken to solve the cube

        """

        while True:

            status = self.search(state, 1)

            if status: return self.moves

            self.moves = []

            self.threshold = self.min\_threshold

        return []

    def search(self, state, g\_score):

        """

        Input: state - string representing the current state of the cube

               g\_score - integer representing the cost to reach the current node

        Description: search the game tree using the IDA\* algorithm

        Output: boolean representing if the cube has been solved

        """

        cube = RubiksCube(state=state)

        if cube.solved():

            return True

        elif len(self.moves) >= self.threshold:

            return False

        min\_val = float('inf')

        best\_action = None

        for a in [(r, n, d) for r in ['h', 'v', 's'] for d in [0, 1] for n in range(cube.n)]:

            cube = RubiksCube(state=state)

            if a[0] == 'h':

                cube.horizontal\_twist(a[1], a[2])

            elif a[0] == 'v':

                cube.vertical\_twist(a[1], a[2])

            elif a[0] == 's':

                cube.side\_twist(a[1], a[2])

            if cube.solved():

                self.moves.append(a)

                return True

            cube\_str = cube.stringify()

            h\_score = self.heuristic[cube\_str] if cube\_str in self.heuristic else self.max\_depth

            f\_score = g\_score + h\_score

            if f\_score < min\_val:

                min\_val = f\_score

                best\_action = [(cube\_str, a)]

            elif f\_score == min\_val:

                if best\_action is None:

                    best\_action = [(cube\_str, a)]

                else:

                    best\_action.append((cube\_str, a))

        if best\_action is not None:

            if self.min\_threshold is None or min\_val < self.min\_threshold:

                self.min\_threshold = min\_val

            next\_action = choice(best\_action)

            self.moves.append(next\_action[1])

            status = self.search(next\_action[0], g\_score + min\_val)

            if status: return status

        return False

def build\_heuristic\_db(state, actions, max\_moves = 20, heuristic = None):

    """

    Input: state - string representing the current state of the cube

           actions -list containing tuples representing the possible actions that can be taken

           max\_moves - integer representing the max amount of moves alowed (default = 20) [OPTIONAL]

           heuristic - dictionary containing current heuristic map (default = None) [OPTIONAL]

    Description: create a heuristic map for determining the best path for solving a rubiks cube

    Output: dictionary containing the heuristic map

    """

    if heuristic is None:

        heuristic = {state: 0}

    que = [(state, 0)]

    node\_count = sum([len(actions) \*\* (x + 1) for x in range(max\_moves + 1)])

    with tqdm(total=node\_count, desc='Heuristic DB') as pbar:

        while True:

            if not que:

                break

            s, d = que.pop()

            if d > max\_moves:

                continue

            for a in actions:

                cube = RubiksCube(state=s)

                if a[0] == 'h':

                    cube.horizontal\_twist(a[1], a[2])

                elif a[0] == 'v':

                    cube.vertical\_twist(a[1], a[2])

                elif a[0] == 's':

                    cube.side\_twist(a[1], a[2])

                a\_str = cube.stringify()

                if a\_str not in heuristic or heuristic[a\_str] > d + 1:

                    heuristic[a\_str] = d + 1

                que.append((a\_str, d+1))

                pbar.update(1)

    return heuristic

*UI*

**HTML**

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta http-equiv="X-UA-Compatible" content="IE=edge">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

    <link rel="stylesheet" href="ui.css">

</head>

<body>

    <div class="textarea">

        <p>Enter a string</p>

        <textarea name="" id="" cols="30" rows="10"></textarea>

        <button onclick="getValue()">submit</button>

    </div>

    <!-- <div class="cube">

        <div class="row center">

            <div class="side top">

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

            </div>

        </div>

        <div class="row">

            <div class="side left">

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

            </div>

            <div class="side front">

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

            </div>

            <div class="side right">

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

            </div>

            <div class="side back">

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

            </div>

        </div>

        <div class="row center">

            <div class="side bottom">

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

            </div>

        </div>

    </div> -->

    <div class="moves">

    </div>

    <script src="https://cdnjs.cloudflare.com/ajax/libs/axios/1.2.1/axios.min.js"></script>

    <script src="./ui.js"></script>

</body>

</html>

**CSS**

:root{

    --box-size : 200px;

}

\*{

    box-sizing: border-box;

    border-collapse: collapse;

}

.cube{

    margin-top: 50px;

    border-top: solid black 2px;

    padding-top: 50px;

}

.cube .move{

    font-size: 32px;

    font-weight: 550;

    text-align: center;

}

.cube .row{

    display: flex;

}

.cube .row.center{

    margin-left: 200px;

}

.side{

    width: var(--box-size);

    height: var(--box-size);

    display: flex;

    flex-wrap: wrap;

    border-collapse: collapse;

}

.side .box{

    width: 33.33%;

    height: 33.33%;

    border: solid black 1px;

    border-collapse: collapse;

}

.box.green{

    background-color: green;

}

.box.red{

    background-color: red;

}

.box.yellow{

    background-color: yellow;

}

.box.orange{

    background-color: orange;

}

.box.blue{

    background-color: blue;

}

**JS (script) file**

let state = 'ooowwwwbwyooggobbbgwggrgyworrwrbbgggbbbyoyrrwyoryyryyr';

let colors = {

    w: 'white',

    o : 'orange',

    y : 'yellow',

    r: 'red',

    b: 'blue',

    g: 'green'

}

function completeSide(str, sidePos, cube){

    let side = str.split('');

    let sideToFill = cube.querySelector(`.${sidePos}`);

    let strToFill = `

                <div class="box ${colors[side[0]]}"></div>

                <div class="box ${colors[side[1]]} "></div>

                <div class="box  ${colors[side[2]]}"></div>

                <div class="box  ${colors[side[3]]}"></div>

                <div class="box  ${colors[side[4]]}"></div>

                <div class="box  ${colors[side[5]]}"></div>

                <div class="box  ${colors[side[6]]}"></div>

                <div class="box  ${colors[side[7]]}"></div>

                <div class="box  ${colors[side[8]]}"></div>

    `

    sideToFill.innerHTML = strToFill;

}

function fillCubes(cube, str){

    let order = ['top', 'left', 'front', 'right', 'back', 'bottom'];

    let sideStrings = ['','','','','',''];

    let sideStringsIndex = 0;

    sideStrings[0] = '';

    for (var i = 0; i < str.length; i++) {

        if(i > 0 && i%9 == 0){

            sideStringsIndex ++;

        }

        sideStrings[sideStringsIndex] += str.charAt(i)

        completeSide(sideStrings[sideStringsIndex], order[sideStringsIndex], cube)

    }

}

function createCube(state, move){

    const cube = document.createElement("div");

    cube.classList.add('cube')

    cube.innerHTML = `

    <div class="row center">

            <div class="side top">

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

            </div>

        </div>

        <div class="row">

            <div class="side left">

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

            </div>

            <div class="side front">

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

            </div>

            <div class="side right">

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

            </div>

            <div class="side back">

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

            </div>

        </div>

        <div class="row center">

            <div class="side bottom">

                <div class="box "></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

                <div class="box"></div>

            </div>

        </div>

    `

    fillCubes(cube, state)

    document.querySelector('body').innerHTML += cube.outerHTML

}

function movesfunc(moves){

    console.log('moves function');

    let movesDiv = document.querySelector('.moves')

    console.log(movesDiv);

    console.log(moves);

    const movesData = {

        h: 'horizontal twist',

        s: 'side twist',

        v: 'vertical twist'

    }

    for(let i = 0; i < moves.length; i++){

        console.log('inside for loo')

        console.log(i);

        let move = moves[i];

        let moveDirection = '';

        let change = '';

        if(move[0] == "v"){

            change = 'colum';

            if(move[2] == 0)moveDirection = 'down';

            if(move[2] == 1)moveDirection = 'up'

        }

        if(move[0] == 'h'){

            change = 'row';

            if(move[2] == 0)moveDirection = 'anticlockwise'

            if(move[2] == 1)moveDirection = 'clockwise'

        }

        if(move[0] == 's'){

            change = 'face';

            if(move[2] == 0)moveDirection = 'anticlockwise'

            if(move[2] == 1)moveDirection = 'clockwise'

        }

        let str = `<p class="move">${movesData[move[0]]} ${change} ${moveDirection} ${move[1]}</p>`

        console.log(str);

        movesDiv.innerHTML += str;

    }

}

async function getStates(stringInp){

    console.log('getting states')

    let data = await axios.post('http://127.0.0.1:5000/', {string: stringInp})

    console.log(data);

    data.data.states.forEach((state, index)=>{

        console.log('creating cube')

        createCube(state);

    })

    movesfunc(data.data.moves);

}

function getValue(){

    console.log('button clicked')

    let text = document.querySelector('textarea').value;

    console.log(text);

    getStates(text)

}